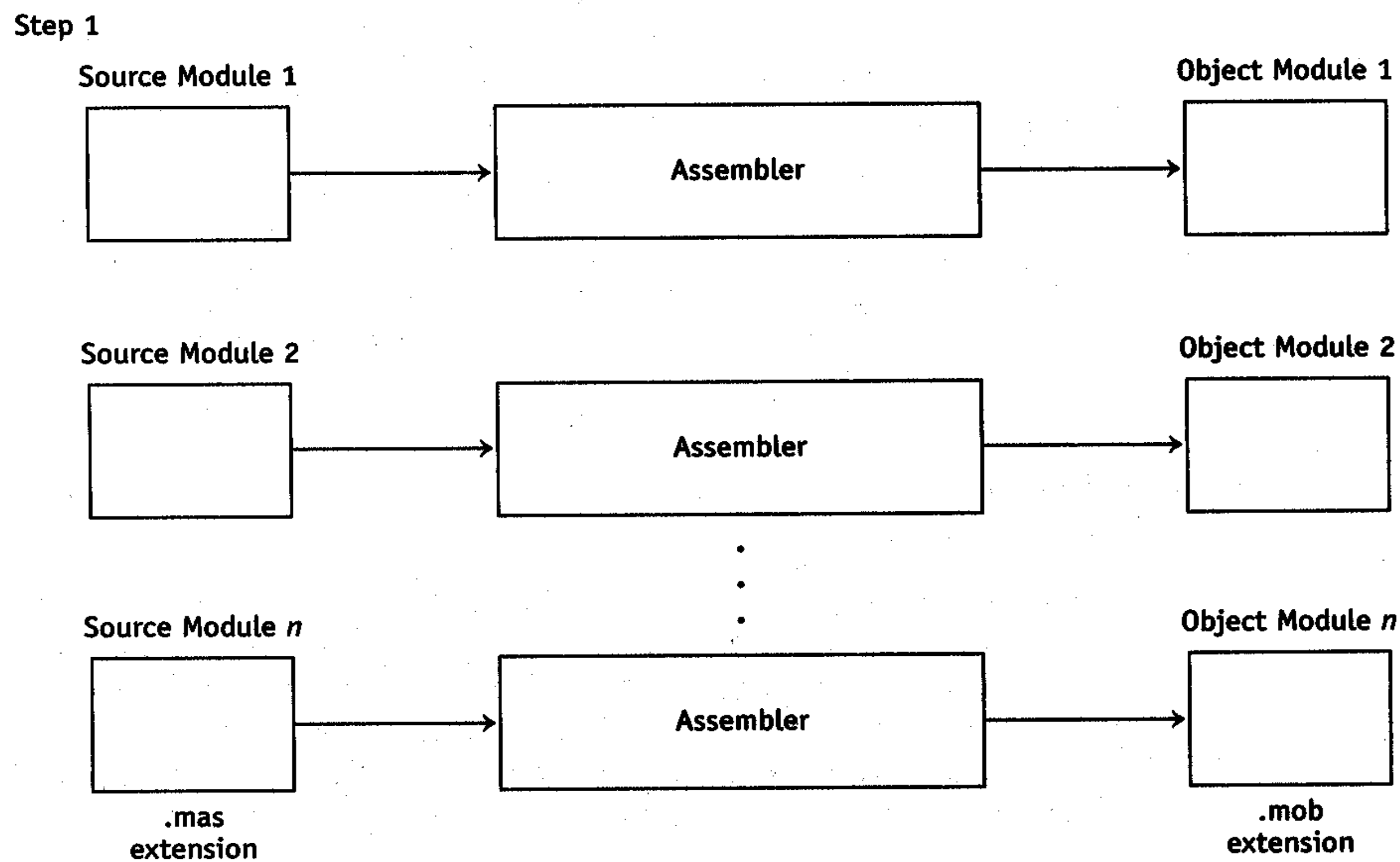
Chapter 10

Linking and Loading

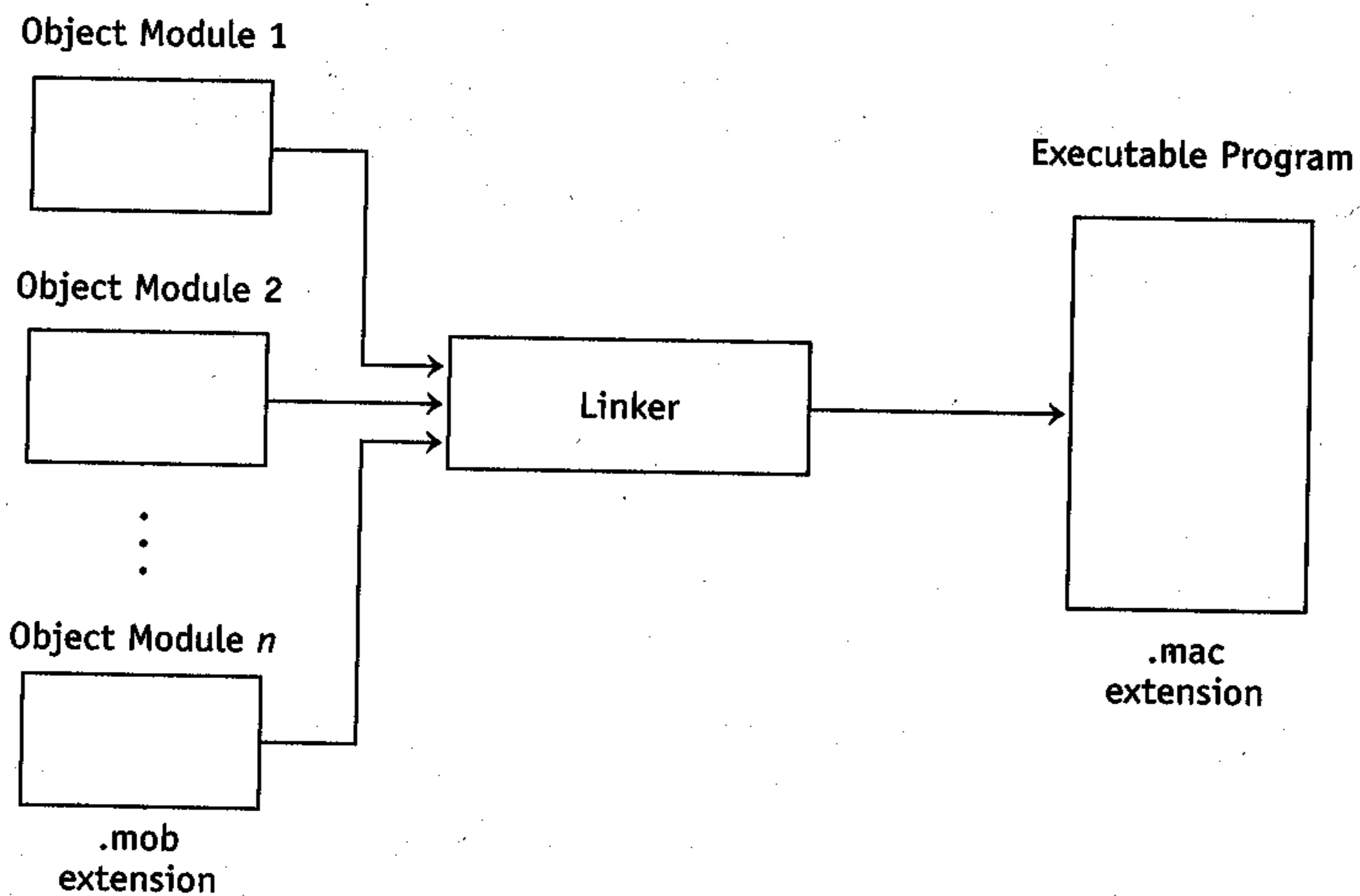
Separate assembly creates “.mob” files

FIGURE 10.1



Linking separately-assembled modules creates a ".mac" file

FIGURE 10.2 Step 2



What if load point = 0? Or 500?

FIGURE 10.3

```
LOC      OBJ              SOURCE
hex*dec
0  *0    9004             ja    add
1  *1    0001  x:         dw    1
2  *2    0008  y:         dw    8
3  *3    0000  z:         dw    0
4  *4    0001  add:       ld    x
5  *5    2002             add   y
6  *6    1003             st    z
7  *7    FFFF             halt
8  *8 ==================== end of program ====================
```

Correct addresses depend on the load point

FIGURE 10.4

Version 1 load point = 0		Version 2 load point = 500 hex		Assembly Code		
Loc		Loc				
0	9**004**	500	9**504**		ja	start
1	0001	501	0001	x:	dw	1
2	0008	502	0008	y:	dw	8
3	0000	503	0000	z:	dw	0
4	0**001**	504	0**501**	start:	ld	x
5	2**002**	505	2**502**		add	y
6	1**003**	506	1**503**		st	z
7	FFFF	507	FFFF		halt	

Correct addresses depend on the load point

FIGURE 10.4

Version 1		Version 2		Assembly Code		
load point = 0		load point = 500 hex				
Loc		Loc				
0	9**004**	500	9**504**		ja	[redacted] add
1	0001	501	0001	x:	dw	1
2	0008	502	0008	y:	dw	8
3	0000	503	0000	z:	dw	0
4	0**001**	504	0**501**	add [redacted]:	ld	x
5	2**002**	505	2**502**		add	y
6	1**003**	506	1**503**		st	z
7	FFFF	507	FFFF		halt	

Use the /p argument to specify the load point

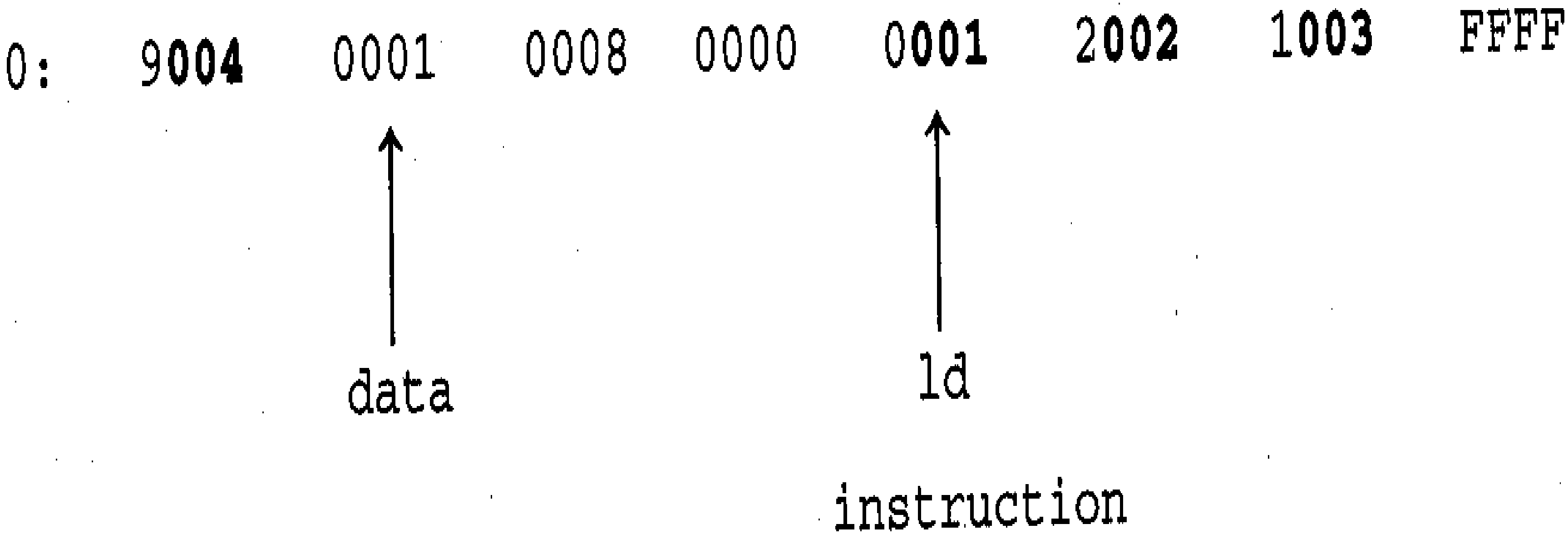
```
sim /p500
```

Program works when load point = 0

FIGURE 10.5

```
C:\sim>sim fig1003   <------ load point defaults to 0
Simulator Version x.x

Starting session.  Enter h or ? for help.
---- [T7] 0: ja   /9 004/ d*
  0:  9004  0001  0008  0000  0001  2002  1003  FFFF        ........
---- [T7] 0: ja   /9 004/ g
  0: ja    /9 004/ pc=0001/0004
  4: ld    /0 001/ ac=0000/0001
  5: add   /2 002/ ac=0001/0009
  6: st    /1 003/ m[003]=0000/0009
  7: halt  /FFFF /
Machine inst count =       5 (hex) =     5 (dec)
---- [T7] q
```

Addresses are in bold



The st instruction correctly stores 9 into location 3 (z).

```
6: st    /1 003/ m[003]=0000/0009
```

Now let's see if the same program works when the load point is 500. Will it have the correct addresses?

```
sim fig1003 /p500
```

It works!

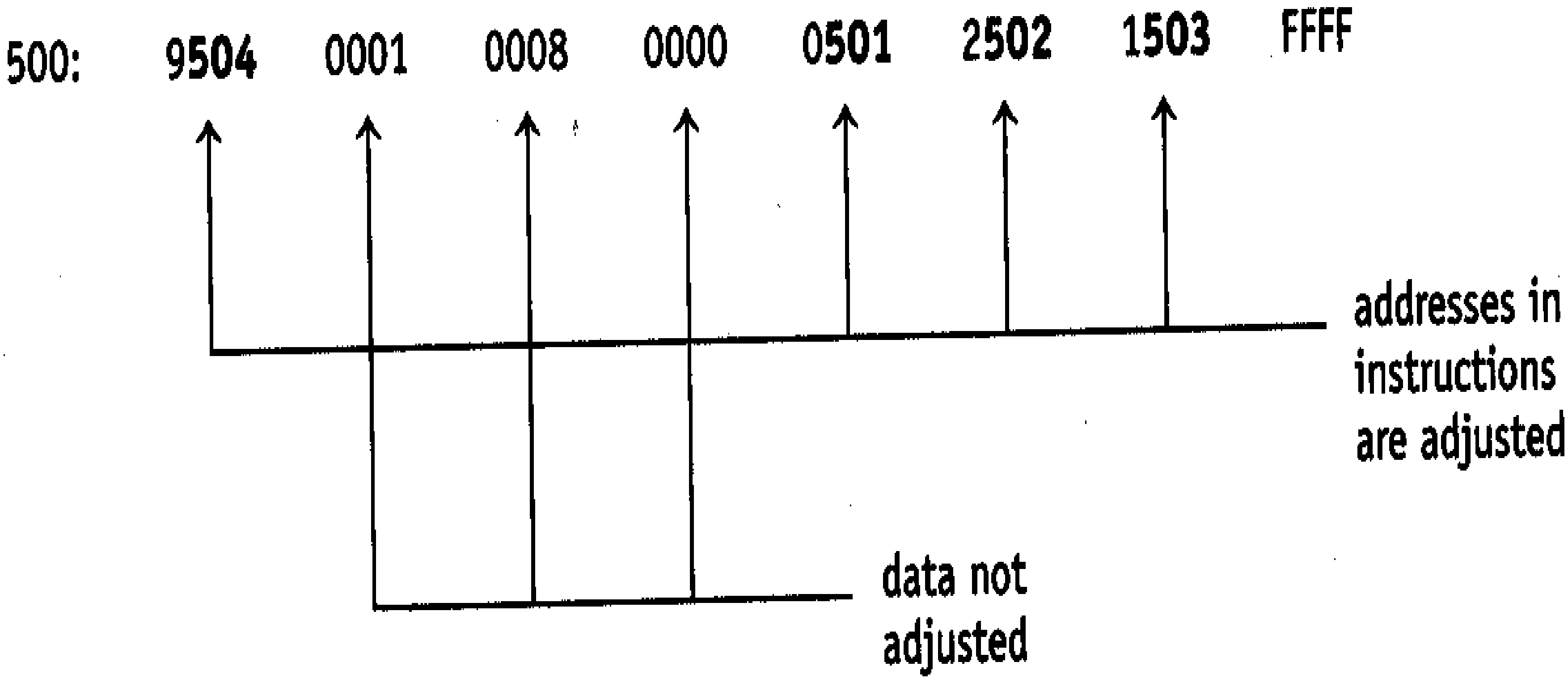
FIGURE 10.6

```
C:\H1>sim /p500    <---- load point is 500 hex
Simulator Version x.x

Starting session.   Enter h or ? for help.
---- [T7] 500: ja   /9 504/ d*

500:  9504  0001  0008  0000  0501  2502  1503  FFFF ........
---- [T7] 500: ja   /9 504/ g

500: ja   /9 504/ pc=0501/0504
504: ld   /0 501/ ac=0000/0001
505: add  /2 502/ ac=0001/0009
506: st   /1 503/ m[503]=0000/0009
507: halt /FFFF /

Machine inst count =   5 (hex) =      5 (dec)
---- [T7] q
```

The st instruction correctly stores 9 into location 503 (the location of z when the load point is 500).

```
506: st    /1 503/ m[503]=0000/0009
```

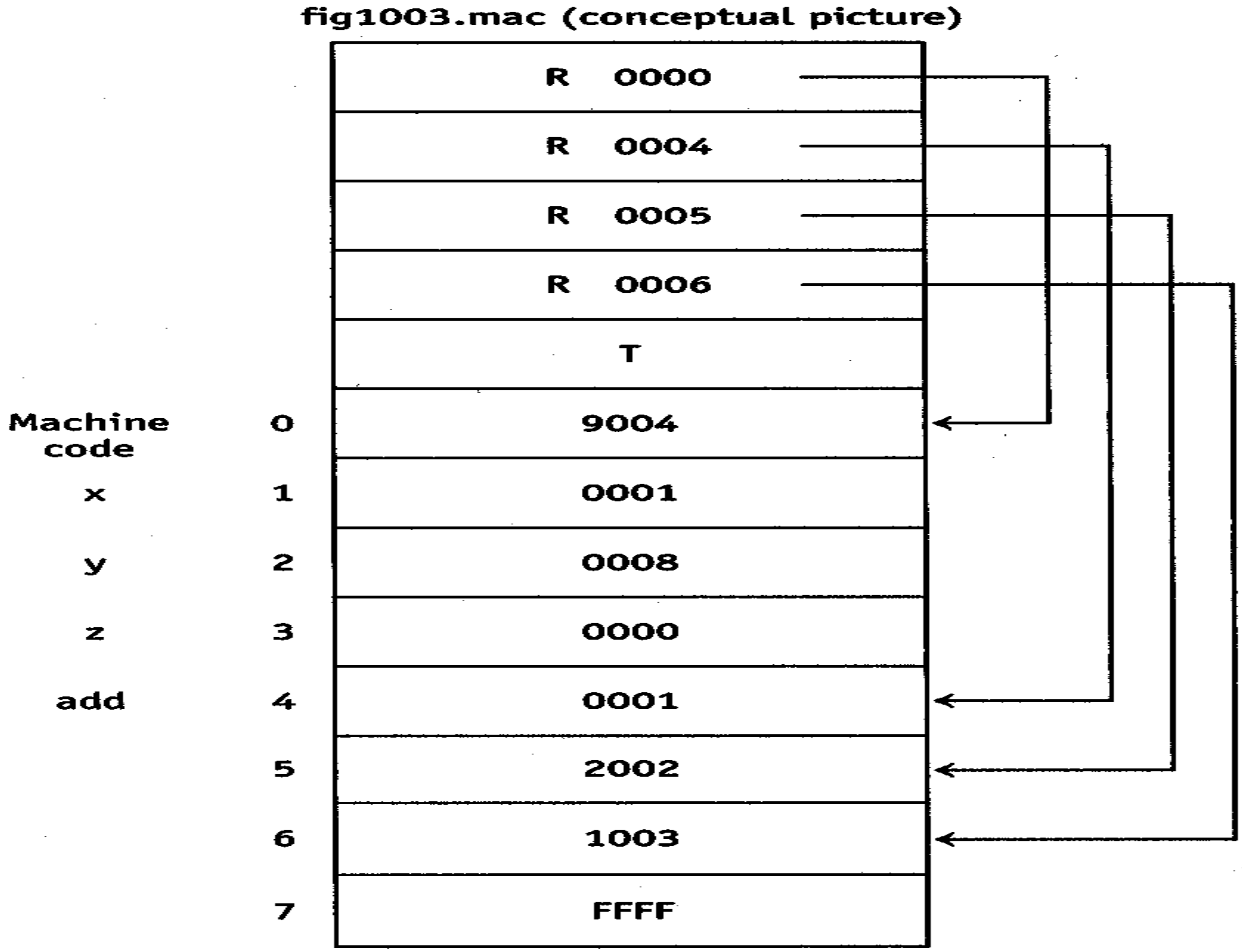
Somehow addresses (and ONLY addresses) were changed to reflect the load point 500. How did **sim** (the OS) know which fields to adjust?



A “.mac” file consists of two parts: the header and the text. The *header* contain R entries which indicate the location of the “relocatable” fields. The *text* is the machine code. A T entry separates the header from the text.

R entries in ".mac" field indicate fields that have to be adjusted.

FIGURE 10.7



How the OS determines the location of the relocatable fields in the loaded program

the location of the beginning of the machine code text in memory (i.e., the load point)

+

the pointer value in the R entry

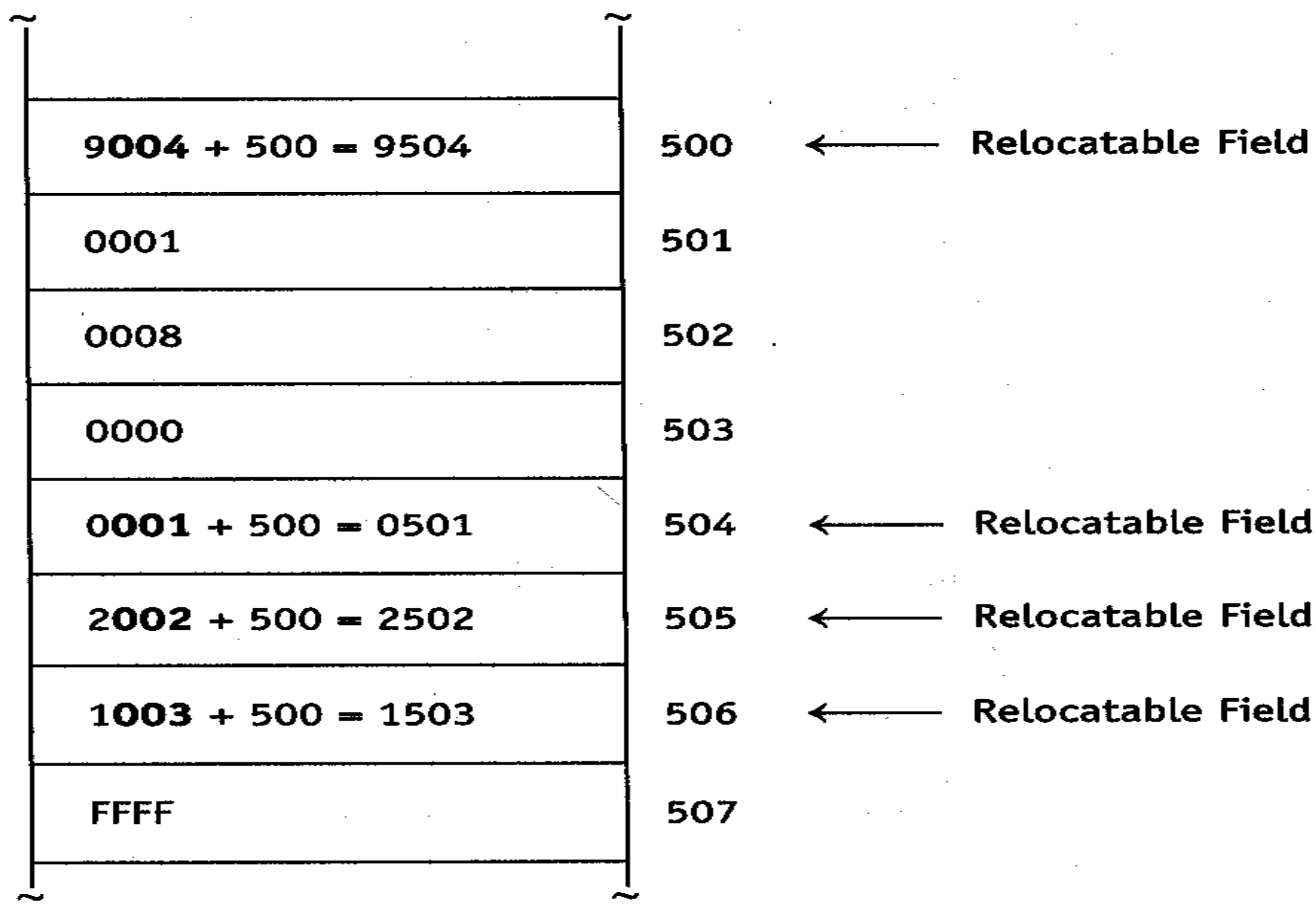
Load Pt   Pointer value in R entries

Load Pt		Pointer value in R entries		
500	+	0	=	500
500	+	4	=	504
500	+	5	=	505
500	+	5	=	506

locations of relocatable fields

OS adds the load point to the relocatable fields of the program as it sits in memory.

FIGURE 10.8



The pic program provides a conceptual picture of a ".mac" file

```
pic fig1003.mac
```

FIGURE 10.9 Conceptual Picture Version x.x

```
Header:

          Type       Address      Symbol
           R          0000
           R          0004
           R          0005
           R          0006
           T

Text:

          Loc        Text         Symbol
           0         9004
           1         0001
           2         0008
           3         0000
           4         0001
           5         2002
           6         1003
           7         FFFF

Input file   = fig1003.mac
List file    = fig1003.pic
```

mex program shows hex and ASCII

FIGURE 10.10

```
Module Examiner Version x.x
Header:

     0:    0052   0000   0052   0004   0052   0005   0052   0006   R.R.R.R.
     8:    0054                                                    T

Text:
     0:    9004   0001   0008   0000   0001   2002   1003   FFFF   ........

Input file = fig1003.mac
List file  = fig1003.mex
```

Don't jump over data. Use end directive instead.

FIGURE 10.11

```
LOC       OBJ    SOURCE
hex*dec

0   *0    0001   x:     dw    1
1   *1    0008   y:     dw    8
2   *2    0000   z:     dw    0
3   *3    0000   add:   ld    x
4   *4    2001          add   y
5   *5    1002          st    z
6   *6    FFFF          halt
                        end   add
7   *7 ================= end of program ==============
```

The end directive results in a small-s entry in the header which indicates the entry point relative to the beginning of the module.

FIGURE 10.12

Type	Address	Symbol
R	0003	
R	0004	
R	0005	
s	0003	← small-s entry
T		

Absolute addresses should not be relocated. An absolute address in the end directive creates a big-S entry in the header.

FIGURE 10.13

```
LOC        OBJ     SOURCE
hex*dec
0   *0     0001    x:      dw    1
1   *1     0008    y:      dw    8
2   *2     0000    z:      dw    0
3   *3     0000    add:    ld    0
4   *4     2001            add   1
5   *5     1002            st    2
6   *6     FFFF            halt
                           end   3
7   *7 ====================== end of program ===============
```

This is the header for the program in the preceding slide. A big-S entry means that the entry point is absolute—the entry point does not depend on the load point. The lack of R entries means no relocation of addresses will occur.

Type	Address	Symbol
S	0003	
T		

Programs consisting of multiple modules are generally better than programs consisting of one big module. With the multiple module approach, the modules need some way to communicate with each other if they are assembled separately.

The two modules in this example are not assembled separately so communication is easy.

FIGURE 10.15

```
LOC      OBJ    SOURCE
hex*dec

0   *0   E004   main:     call   sub       ; call other module sub
1   *1   1008             st     total     ; total is in sub
2   *2   FFFF             halt
3   *3   0011   x:        dw     17
                          end    main

                ;=========================================
4   *4   0003   sub:      ld     x         ; x is in main
5   *5   2007             add    y
6   *6   F000             ret
7   *7   0022   y:        dw     34
8   *8   0000   total:    dw     0
9   *9   ========== end of fig1015.mas ===========================
```

Separate assembly results in assembly-time errors

FIGURE 10.16

m1.mas

```
main:   call   sub
        st     total
done:   halt
x:      dw     17
        end    main
```

Module 1

m2.mas

```
sub:    ld     x
        add    y
done:   ret
y:      dw     34
total:  dw     0
```

Module 2

For separate assembly, we need the public and extern directives.

public makes a symbol global.

e*xtern* indicates that a symbol is defined in some other module.

Now separate assembly works

FIGURE 10.17

m1.mas

```
main:  call    sub
       st      total
done:  halt
x:     dw      17
       public  x
       extern  sub
       extern  total
       end     main
```

Module 1

m2.mas

```
sub:   ld      x
       add     y
done:  ret
y:     dw      34
total: dw      0
       public  sub
       public  total
       extern  x
```

Module 2

The assembler does not know the addresses of the external symbols *sub* and *total* when it is assembling *m1*, so it uses 0 for their addresses.

```
call sub
st    total
```

is translated to

```
E000
1000
```

Assembly listings

FIGURE 10.18

Module 1

```
  LOC      OBJ     SOURCE
hex*dec

0   *0     E000    main:   call   sub
1   *1     1000            st     total
2   *2     FFFF    done:   halt
3   *3     0011    x:      dw     17
                           public x
                           extern sub
                           extern total
                           end    main

Input file   = m1.mas
Output file = m1.mob
```

Module 2

```
LOC      OBJ     SOURCE
hex*dec

0   *0   0000   sub:     ld      x
1   *1   2003            add     y
2   *2   F000   done:    ret
3   *3   0000   y:       dw      34
4   *4   0022   total:   dw      0
                         public sub
                         public total
                         extern x

Input file   = m2.mas
Output file = m2.mob
```

When the linker combines m1 and m2, It must update addresses. For example, references to total must contain the new address of total, computed as follows:

4	(the starting address of module 2 in the combined program)
+ 4	(the address of `total` relative to the beginning of module 2)
8	(the address of `total` in the combined program)

The linker must update the address in the add instruction:

```
   3 (address in add instruction put there by the assembler)
+  4 (address at which module 2 starts in the combined program)
----
   7 (new address in the add instruction—the address of y
     in the combined program)
```

To assemble and link:

mas m1
mas m2
lin m1 m2

lin creates m1.mac

To see “.mob” and “.mac” files use the **pic** or **mex** programs.

pic m1.mob

mex m1.mob

pic m2.mob

mex m2.mob

pic m1.mac

mex m1.mac

pic output for m1.mob and m2.mob
(‘^’ flags an external symbol)

FIGURE 10.19

```
Header:
        Type    Address    Symbol
         E       0000       sub
         E       0001       total
         P       0003       x
         S       0000
         T

Text:
      Address   Text       Symbol
         0      E000       ^sub
         1      1000       ^total
         2      FFFF
         3      0011       x

Input file    = m1.mob
List file     = m1.pic
```

```
Header:
        Type    Address    Symbol
         E       0000       x
         R       0001
         P       0000       sub
         P       0004       total
         T

Text:
      Address   Text       Symbol
         0      0000       sub^x
         1      2003
         2      F000
         3      0022
         4      0000       total

Input file    = m2.mob
List file     = m2.pic
```

mex output for m1.mob

FIGURE 10.20

```
Module Examiner Version x.x
Header:
         0:   0045  0000  0073  0075  0062  0000  0045  0001    E.sub.E.
         8:   0074  006F  0074  0061  006C  0000  0050  0003    total.P.
        10:   0078  0000  0073  0000  0054                      x.s.T
Text:
         0:   E000  1000  FFFF  0011                            ....
Input file = m1.mob
List file  = m1.mex
```

There is an E entry in a header for every external reference

```
extern t        ; only 1 extern directive
ld   t          ; first external reference to t
add  t          ; second external reference to t
st   t          ; third external reference to t
halt
```

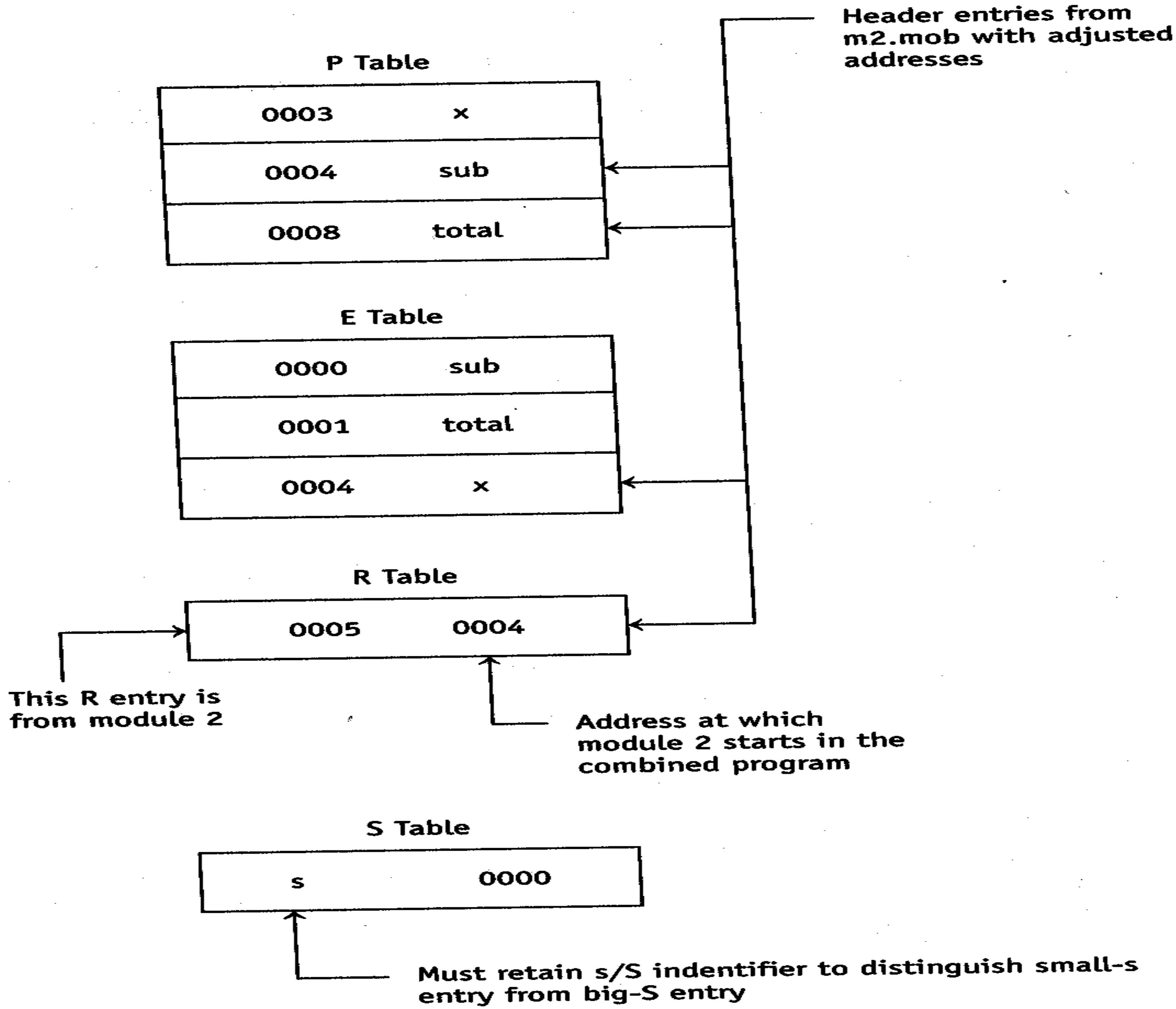
The three E entries are

```
E    0000    t
E    0001    t
E    0002    t
```

Step 1 in linking process: read in modules to be linked.

- Addresses in header entries are adjusted.
- The modified header entries are saved in internal tables (P table, E table, R table, and S table). When an R entry is saved, the start address of the associated module is included in the saved R entry.
- The text portions of the modules are loaded one after the other into a text buffer.

FIGURE 10.21 Saved Header Entries



Step 2 in the linking process: resolve external references. For each E entry,

- Determine the address in the combined program of the external field (this address is in the saved E entry).
- By searching the P entries, determine the location in the combined program of the symbol referenced.
- Add the address of the symbol referenced to the external field.

Step 3 in the linking process: relocate relocatable fields. For each R entry,

- Get the address of the relocatable field from the R entry.
- Add the module start address (also obtained from the R entry) to the relocatable field in the combined text.

Example of R entry processing

R   0005   0004

Add 0004 to the relocatable field in the combined text at location 5 from the start of the text.

FIGURE 10.22

a) Before resolution of external references

Before relocation of addresses

Text for Combined Program

main	0	E000	External reference to sub (location 4)
	1	1000	External reference to total (location 8)
	2	FFFF	
x	3	0011	
sub	4	0000	External reference to x (location 3)
	5	2003	Contains address that needs relocation
	6	F000	
y	7	0022	
total	8	0000	

b) After resolution of external references

Before relocation of addresses

Text for Combined Program

main	0	E004	Resolved external reference to sub
	1	1008	Resolved external reference to total
	2	FFFF	
x	3	0011	
sub	4	0003	Resolved external reference to x
	5	2003	Contains address that needs relocation
	6	F000	
y	7	0022	
total	8	0000	

c) After resolution of external references

After relocation of addresses

Text for Combined Program

main	0	E004	Resolved external reference to sub
	1	1008	Resolved external reference to total
	2	FFFF	
x	3	0011	
sub	4	0003	Resolved external reference to x
	5	2007	Contains adjusted address
	6	F000	
y	7	0022	
total	8	0000	

Last step in the linking process: output the “.mac” file.

- The saved headers, with all E entries turned into R entries, are outputed.
- The combined text in the text buffer (which now has all its addresses adjusted appropriately) is outputed.

pic output for m1.mac

FIGURE 10.23

a) **pic** Display

```
Conceptual Picture Version x.x

Header:
            Type        Address      Symbol
             P           0003        x
             P           0004        sub
             P           0008        total
             R           0005
             R           0000
             R           0001
             R           0004
             S           0000
             T

Text:
          Address       Text         Symbol
             0           E004
             1           1008
             2           FFFF
             3           0011        x
             4           0003        sub
             5           2007
             6           F000
             7           0022
             8           0000        total

Input file  = m1.mac
List file   = m1.pic
```

mex output for m1.mac

```
b) mex Display

Module Examiner Version x.x

Header:
        0:   0050 0003 0078 0000 0050 0004 0073 0075     P.x.P.su
        8:   0062 0000 0050 0008 0074 006F 0074 0061     b.P.tota
       10:   006C 0000 0052 0005 0052 0000 0052 0001     l.R.R.R.
       18:   0052 0004 0073 0000 0054                    R.s.T

Text:
        0:   E004 1008 FFFF 0011 0003 2007 F000 0022     ......."
        8:   0000                                        .

Input file = m1.mac
List file  = m1.mex
======================================================================
```

lin creates a file (with extension ".tab") that shows the link process.

See the ".tab" file on the next slide.

FIGURE 10.24 The third file `lin` creates the *cat file*—a file used by `sim` for source-level tracing.

```
lin Version x.x Table Trace

Tables constructed from user-specified modules:

P Table        Address (hex*dec)          Symbol
               3*3                        x
               4*4                        sub
               8*8                        total

E Table        Address (hex*dec)          Symbol
               0*0                        sub
               1*1                        total
               4*4                        x

R Table        Address (hex*dec)          Module Address (hex*dec)
                     5*5                             4*4

S Table        Address (hex*dec)          Type
                     0*0                   s

================================================================

Text as transformed by E-entry and R-entry processing:

Address        Before E                     After  E      After E
hex*dec        Before R                     Before R      After R

0   *0               E000          change> E004             E004
1   *1               1000          change> 1008             1008
2   *2               FFFF                  FFFF             FFFF
3   *3             x 0011                  0011             0011
4   *4           sub 0000          change> 0003             0003
5   *5               2003                  2003   change> 2007
6   *6               F000                  F000             F000
7   *7               0022                  0022             0022
8   *8         total 0000                  0000             0000

Output file  = m1.mac
Table file   = m1.tab
Cat file     = m1.cat
Label status = case sensitive
```

A *library* is a collection of object modules collected into a single file. Let’s create a library consisting of several simple math modules.

A technique for multiplying n by 15: shift n left 4 times (which multiplies by 16) and then subtract n.

$$16 \times n - n = 15 \times n$$

FIGURE 10.25

```
                                  ; fmult15.mas
mult15:     aloc    1             ; allocate local variable temp
            ldr     2             ; get parameter (n)
            addr    2             ; get 2n
            str     0             ; save 2n in temp
            addr    0             ; get 4n
            str     0             ; save 4n in temp
            addr    0             ; get 8n
            str     0             ; save 8n in temp
            addr    0             ; get 16n
            subr    2             ; get 16n - n = 15n
            dloc    1             ; deallocate local variable temp
            ret                   ; product returned in ac reg
            public mult15
```

To square a whole number n, add the first n odd numbers. For example,

$$3^2 = 1 + 3 + 5 = 9$$

FIGURE 10.26

```
                                        ; fsquare.mas
square:    aloc    2                    ; create local variables sum, odd
           ldc     0
           str     1                    ; initialize sum to 0
           ldc     1                    ; initialize odd to 1
           str     0
loop:      ldr     3                    ; get parameter (count)
           jz      done                 ; all done if count = 0
           sub     @1                   ; subtract 1 from count
           str     3                    ; save count
           ldr     1                    ; get sum
           addr    0                    ; add odd to sum
           str     1                    ; save sum
           ldr     0                    ; get odd
           add     @2                   ; add 2 to odd
           str     0                    ; save odd
           ja      loop
done:      ldr     1                    ; get sum
           dloc    2                    ; deallocate local variables
           ret                          ; square returned in ac reg
@1:        dw      1
@2:        dw      2
           public square
```

Now create a function poly that computes $15 \times n^2$ that calls the square and mult15 modules.

FIGURE 10.27

```
                          ; fpoly.mas
poly:    ldr     1        ; get parameter to poly
         push             ; create parameter to square
         call    square
         dloc    1        ; remove parameter
         push             ; create parameter to mult15
         call    mult15
         dloc    1        ; remove parameter
         ret              ; return result
         public poly
         extern square
         extern mult15
```

Create a main module that calls poly, passing it 2.

main module that calls poly

FIGURE 10.28

```
                                ; fmain.mas
main:   ldc     2               ; pass poly 2
        push
        call    poly
        dloc    1
        dout                    ; output is 15 x 2 squared = 60
        ldc     '\n'
        aout
        halt
        extern poly
```

To create an executable program, first assemble. Use file names—not module names

```
mas fmain
mas fmult15
mas fsquare
mas fpoly
```

Next, link all the modules

```
lin fmain fmult15 fsquare fpoly
```

must be first

It is easy to make an error when linking—you have to remember the names of all the modules that are needed.

```
lin fmain fpoly
```

`lin` would respond with the error message

```
Unresolved external reference square
```

~~reference~~ symbol

Creating a library fmult15.lib

mas fmult15

mas fsquare

mas fpoly

lib fmult15 fsquare fpoly

Base name of library defaults to base name of first file specified.

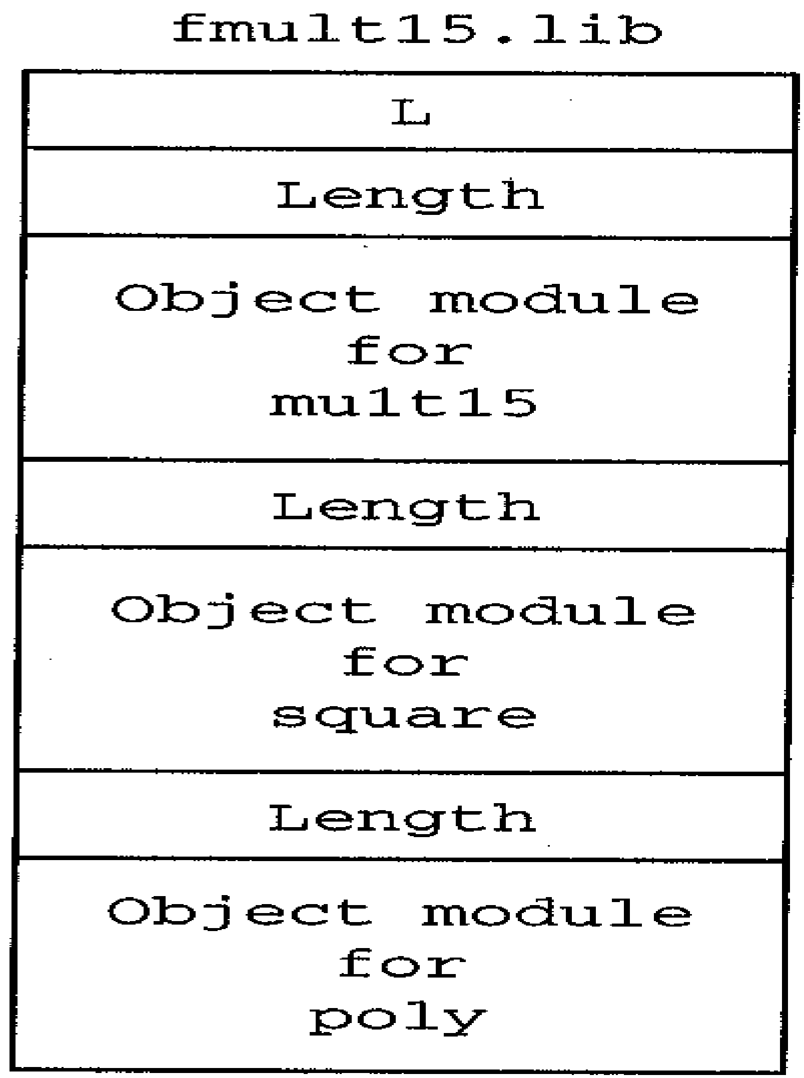
Using the library fmult15.lib

lin fmain /Lfmult15

Now you do not have to remember all the modules that fmain requires.

Library format

FIGURE 10.29



Linking with a library involves five steps. See the next slide.

The `/L` argument specifies the library (`lin` assumes the extension ".lib" for library names). `lin` then performs the following steps:

1. `lin` inputs the specified input files (only `fmain.mob` in this example), adjusting and saving the header entries, and appending the text to the text in its text buffer. (Initially, the text buffer is empty.)
2. For each saved E entry, `lin` performs the following operations:
   a. It gets the external symbol that is stored in the E entry.
   b. It searches among its saved P entries for an entry for this symbol.
   c. If it cannot find the required P entry among its saved entries, it then searches for it in the libraries specified on the command line (`fmult15.lib` in this example). If it finds a module with the required P entry in a library, it inputs this module, adjusting and saving its header entries and appending its text to the text in the text buffer. If, on the other hand, it cannot find the P entry in any library, it generates an "unresolved external reference" error message and terminates.
   d. Assuming `lin` finds the P entry it needs, it resolves the external reference that it is currently processing.
3. `lin` adjusts all the locations in the text that require it, which are those pointed to by the saved R entries. `lin` adjusts these locations by adding to them their corresponding module addresses that also appear in the R table entry (see Figure 10.21).
4. `lin` outputs a header that is appropriate for the final executable program to the output file (`fmain.mac` in this example). This header consists of all the saved header entries, with all the E entries converted to R entries.
5. It outputs the text in the text buffer (which is now resolved and adjusted) to the output file (`fmain.mac` in this example).

Libraries can be displayed with pic and mex. The next slide shows the mex output for the fmult15.lib library.

FIGURE 10.30

```
004C (L for Library)
==================== Module Number 1 ===============================
Length = 0016 (hex) 22 (decimal)

Header:
         0:   0050  0000  006D  0075  006C  0074  0031  0035     P.mult15
         8:   0000  0054                                         .T
Text:
         0:   F501  4002  6002  5000  6000  5000  6000  5000     ........
         8:   6000  7002  F601  F000                             ....
==================== Module Number 2 ===============================
Length = 0027 (hex) 39 (decimal)

Header:
         0:   0052  0006  0052  0007  0052  000D  0052  000F     R.R.R.R.
         8:   0050  0000  0073  0071  0075  0061  0072  0065     P.square
        10:   0000  0054                                         .T
Text:
         0:   F502  8000  5001  8001  5000  4003  C010  3013     ........
         8:   5003  4001  6000  5001  4000  2014  5000  9005     ........
        10:   4001  F602  F000  0001  0002                       .....
==================== Module Number 3 ===============================
Length = 0022 (hex) 34 (decimal)

Header:
         0:   0045  0002  0073  0071  0075  0061  0072  0065     E.square
         8:   0000  0045  0005  006D  0075  006C  0074  0031     .E.mult1
        10:   0035  0000  0050  0000  0070  006F  006C  0079     5.P.poly
        18:   0000  0054                                         .T
Text:
         0:   4001  F300  E000  F601  F300  E000  F601  F000     ........

Input file = fmult15.lib
List file  = fmult15.mex
```

The linker always links entire object modules from a library even if only one function within a module is needed. See the next slide.

f1, f2, and f3 linked even if only f3 needed.

FIGURE 10.31

```
            public  f1
            public  f2
            public  f3
f1:         aloc  5.
            ...
            ret
;====================
f2:         ldc    5
            ...
            ret
;====================
f3:         aloc  1
            ...
            ret
```

Can specify multiple libraries

```
lin z1 z2 /Llib1 /Llib2 /Llib3
```

For a link with a library, **lin**’s “.tab” file shows the loading sequence of modules from the library. See the next slide.

FIGURE 10.32 lin Version x.x Table Trace

```
Tables constructed from user-specified modules:

P Table empty

E Table        Address (hex*dec)    Symbol
                      2*2            poly

R Table empty

S Table empty

================================================================

***** Searching for poly in fmult15.lib library
***** Inputing poly module from fmult15.lib library

================================================================

Tables updated with header information from poly module:

P Table        Address (hex*dec)    Symbol
                      8*8            poly

E Table        Address (hex*dec)    Symbol
                      2*2            poly
                      A*10           square
                      D*13           mult15

R Table empty

S Table empty

================================================================

***** Searching for square in fmult15.lib library
***** Inputing square module from fmult15.lib library

================================================================

Tables updated with header information from square module:

P Table        Address (hex*dec)    Symbol
                      8*8            poly
                      10*16          square

E Table        Address (hex*dec)    Symbol
                      2*2            poly
                      A*10           square
                      D*13           mult15

R Table        Address (hex*dec)    Module Address (hex*dec)
                      16*22               10*16
```

FIGURE 10.32
(continued)

```
                       17*23                  10*16
                       1D*29                  10*16
                       1F*31                  10*16

S Table empty

============================================================

***** Searching for mult15 in fmult15.lib library
***** Inputing mult15 module from fmult15.lib library

============================================================

Tables updated with header information from mult15 module:

P Table        Address (hex*dec)    Symbol
                   8*8               poly
                  10*16              square
                  25*37              mult15

E Table        Address (hex*dec)    Symbol
                  2*2                poly
                  A*10               square
                  D*13               mult15

R Table        Address (hex*dec)    Module Address (hex*dec)
                 16*22                 10*16
                 17*23                 10*16
                 1D*29                 10*16
                 1F*31                 10*16

S Table empty

============================================================

Text as transformed by E-entry and R-entry processing:

Address              Before E          After  E        After E
hex*dec              Before R          Before R        After R

0   *0                 8002              8002            8002
1   *1                 F300              F300            F300
2   *2                 E000     change>  E008            E008
3   *3                 F601              F601            F601
4   *4                 FFFD              FFFD            FFFD
5   *5                 800A              800A            800A
6   *6                 FFFB              FFFB            FFFB
7   *7                 FFFF              FFFF            FFFF
8   *8            poly 0000    change>   4001            4001
9   *9                 0000     change>  F300            F300
A   *10                0000     change>  E010            E010
```

(continued)

FIGURE 10.32 (continued)

```
B   *11           0000       change> F601              F601
C   *12           0000       change> F300              F300
D   *13           0000       change> E025              E025
E   *14           0000       change> F601              F601
F   *15           0000       change> F000              F000
10  *16   square  0000       change> F502              F502
11  *17           0000       change> 8000              8000
12  *18           0000       change> 5001              5001
13  *19           0000       change> 8001              8001
14  *20           0000       change> 5000              5000
15  *21           0000       change> 4003              4003
16  *22           0000       change> C010    change>   C020
17  *23           0000       change> 3013    change>   3023
18  *24           0000       change> 5003              5003
19  *25           0000       change> 4001              4001
1A  *26           0000       change> 6000              6000
1B  *27           0000       change> 5001              5001
1C  *28           0000       change> 4000              4000
1D  *29           0000       change> 2014    change>   2024
1E  *30           0000       change> 5000              5000
1F  *31           0000       change> 9005    change>   9015
20  *32           0000       change> 4001              4001
21  *33           0000       change> F602              F602
22  *34           0000       change> F000              F000
23  *35           0000       change> 0001              0001
24  *36           0000       change> 0002              0002
25  *37   mult15  0000       change> F501              F501
26  *38           0000       change> 4002              4002
27  *39           0000       change> 6002              6002
28  *40           0000       change> 5000              5000
29  *41           0000       change> 6000              6000
2A  *42           0000       change> 5000              5000
2B  *43           0000       change> 6000              6000
2C  *44           0000       change> 5000              5000
2D  *45           0000       change> 6000              6000
2E  *46           0000       change> 7002              7002
2F  *47           0000       change> F601              F601
30  *48           0000       change> F000              F000

Output file  = fmain.mac
Table file   = fmain.tab
Cat file     = fmain.cat
Label status = case sensitive
```

Advantages of separate assembly

- Can control scope of identifiers.
- Saves time to update an executable file—you need to reassemble only those modules that are changed, and then link all the modules.
- Can avoid problems because of assembler limitations, such as symbol table size.
- *Most important*: Can use libraries.

Start-up code

Code that is part of an executable module obtained from a C++ program. Start-up code gets control first from the OS. It handles start-up initialization and then calls **main**.

Suppose you invoke the test program as follows:

```
test a1 a2 a3
```

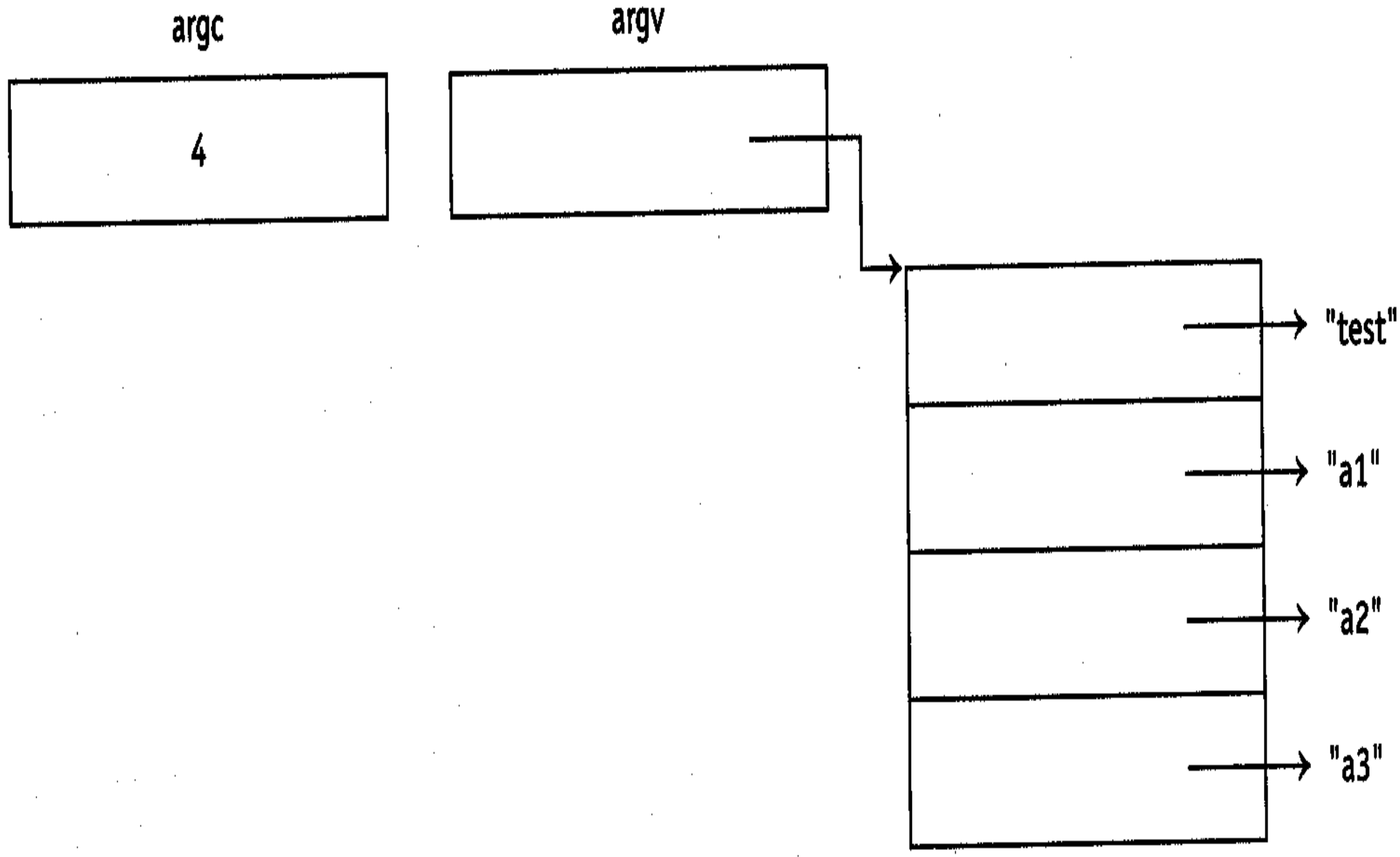
The operating system would then make this command line available to start-up code, which, in turn, would

1. Parse (i.e., break up) the command line into its component parts
2. Build the two arguments shown in Figure 10.33, commonly named `argc` and `argv`, containing information on the command line
3. Call the `main` function, passing it `argc` and `argv`.

`argc` contains the number of items on the command line; `argv` is a pointer to an array of pointers, which, in turn, point to the individual items on the command

Arguments passed to main

FIGURE 10.33



Linking with start-up code sup.mob

Start-up code has an end directive so the order of linking modules is not critical.

```
mas fmain
mas fsub
lin fmain fsum sup
```

FIGURE 10.34

fmain.cpp

```
int main()
{
    . . .
    sub();
    . . .
    return 0;
}
```

fsub.cpp

```
void sub()
{
    . . .
}
```

How to set up a program to use start-up code

FIGURE 10.35

sup.mas

```
; standard start-up code
start_up:
;       initialization code
           . . .
        call main
;       final housekeeping code
           . . .
        extern main
        end start_up
```

fmain.mas

```
; main function
main:   . . .
        call @sub$v
        . . .
        ldc  0
        ret
        public main
        extern @sub$v
```

fsub.mas

```
; sub function
@sub$v: . . .
        . . .
        . . .
        ret
        public @sub$v
```

Suppose the **test** program includes start-up code and it is invoked with

```
sim test a1 a2 a3
```

The next three slides shows how the **argc** and **argv** parameters are built for this invocation.

FIGURE 10.37a

Initial Data Structures

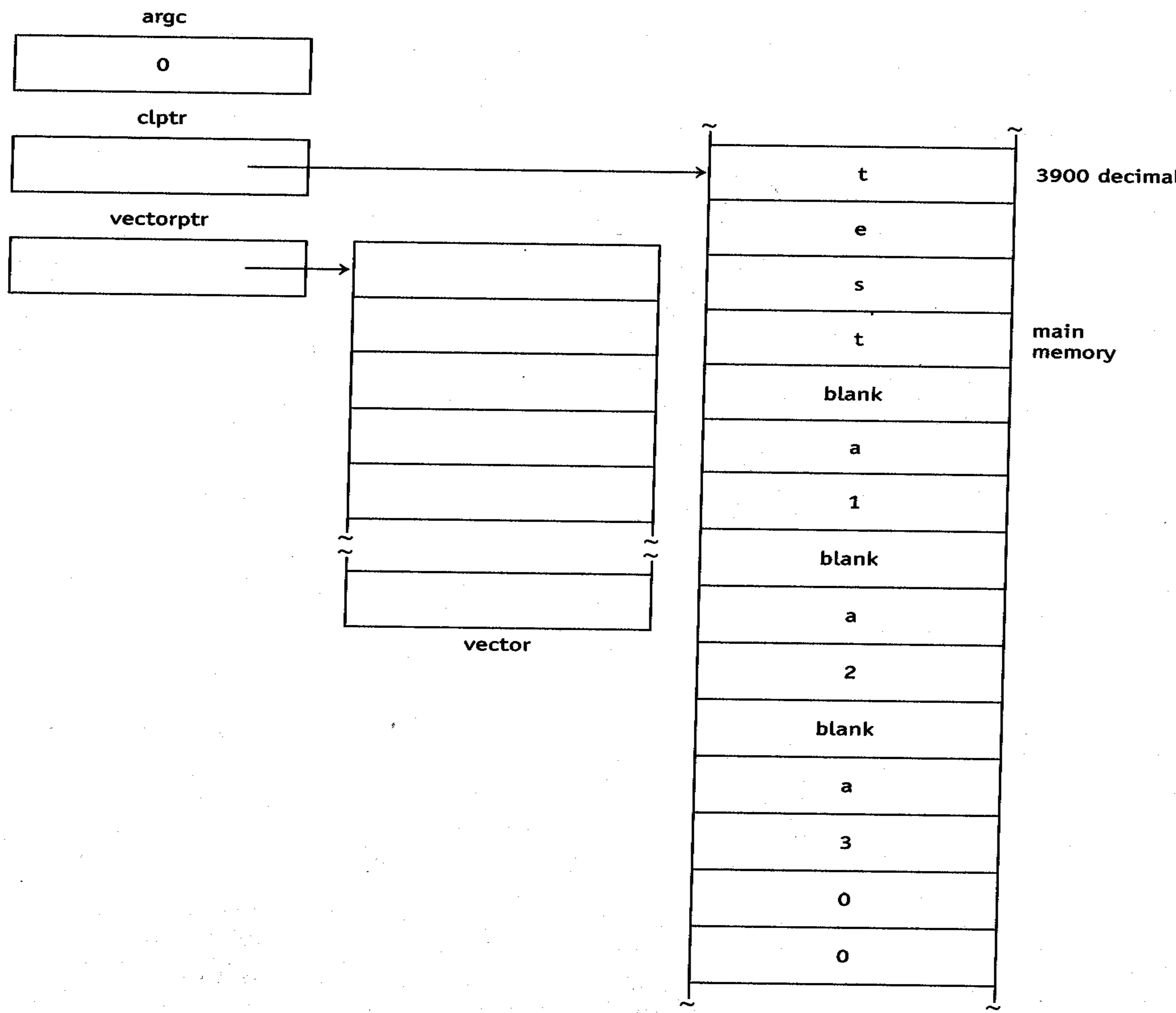


FIGURE 10.37b

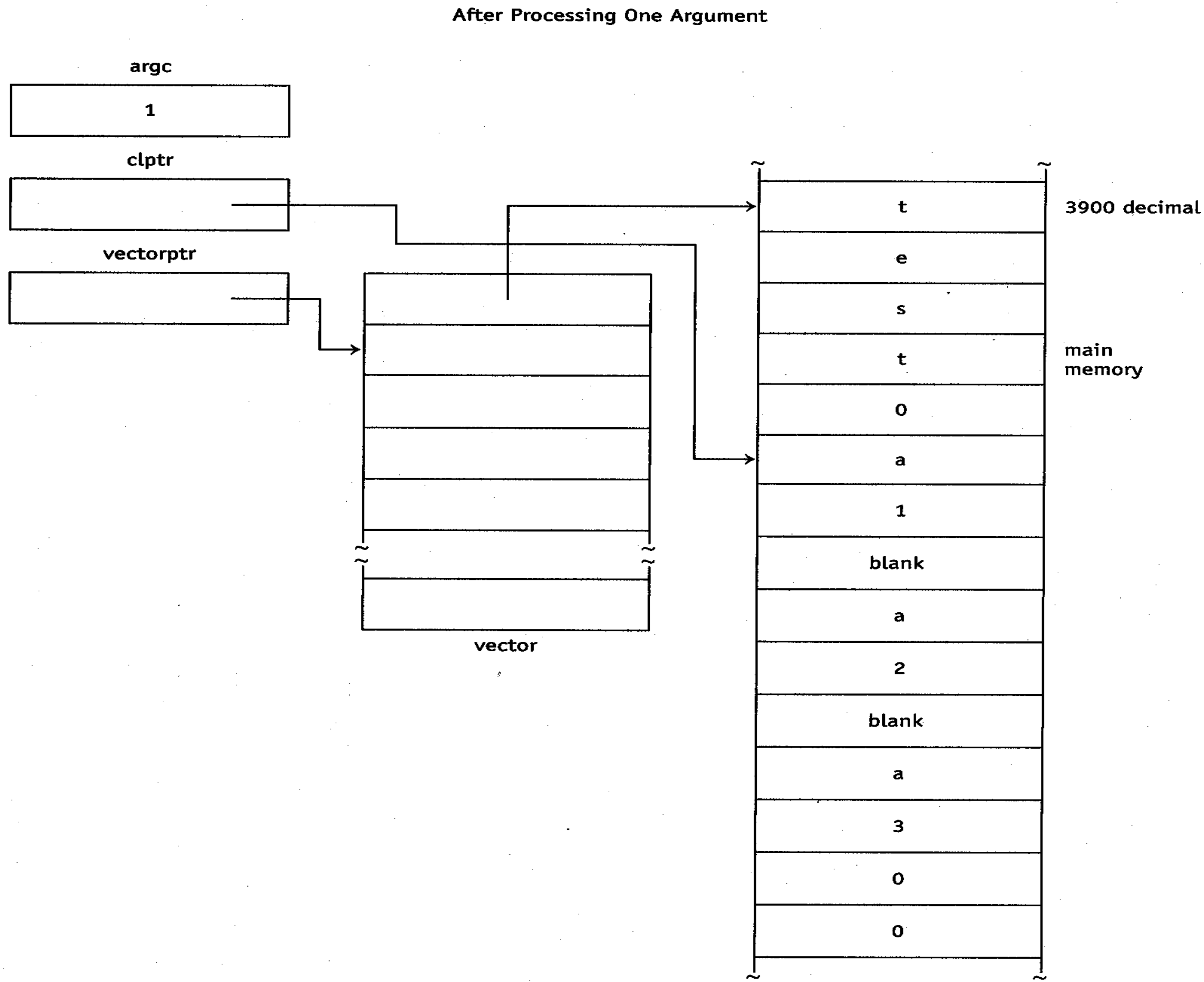


FIGURE 10.37C

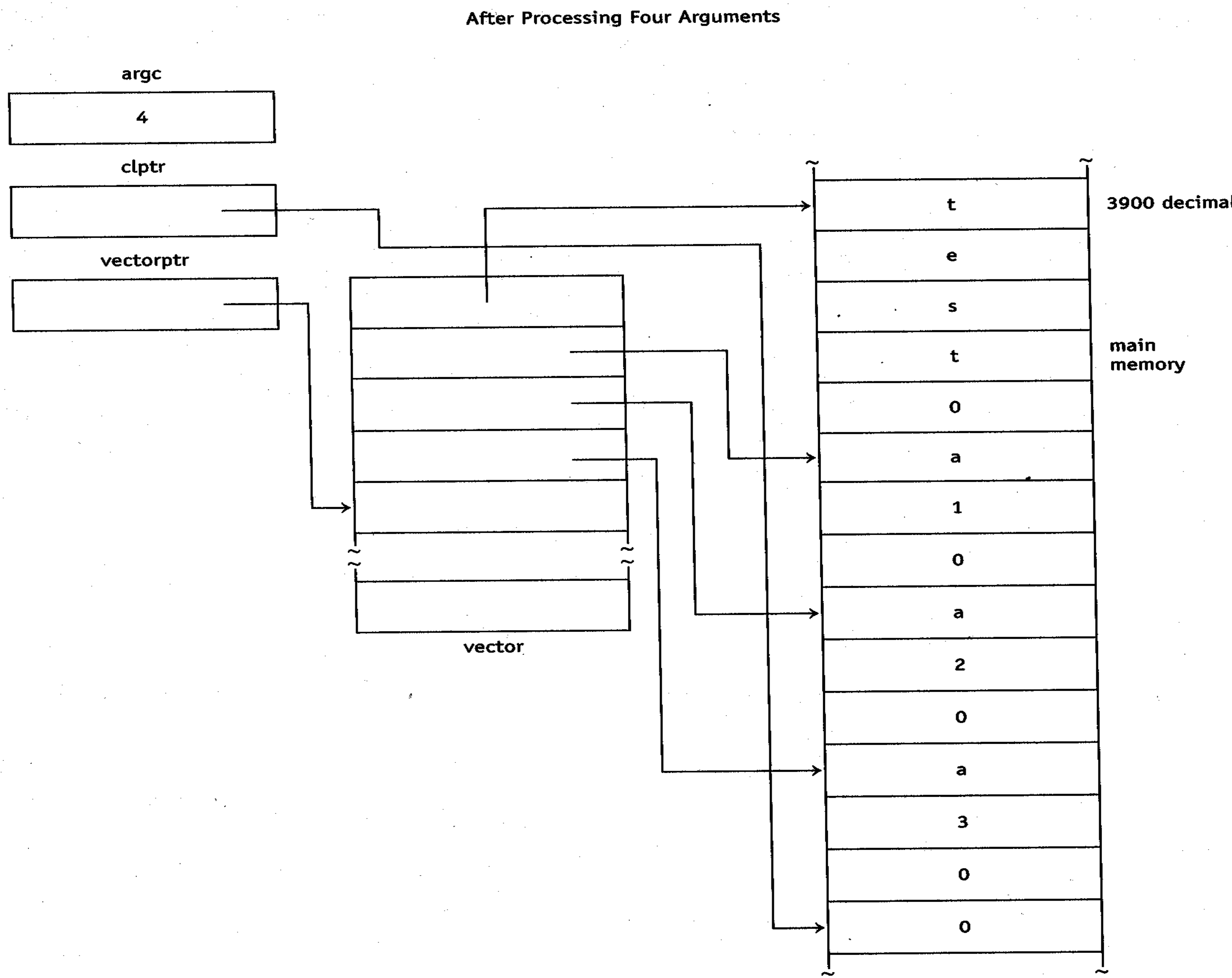


FIGURE 10.36

```
; standard start-up code                                       sup.mas
; ==========================================================================
loc0:         dw    'Z'                     ; to test for null pointer assignment

;             initialize sp register
start_up:     ldc   0
              swap

;             test if clptr has reached the end of the command line
getarg:       ld    clptr                   ; get next char in command line
              ldi
              jz    alldone                 ; if null char, all done

;             check if too many args—max = 20
              ld    argc                    ; get current count
              sub   @20
              jnz   * + 4
              ldc   errmsg1                 ; display error message
              sout
              ja    done                    ; terminate execution if count at 20

;             move contents of clptr into next avail slot in vector
              ld    clptr                   ; get address of next arg
              push
              ld    vectorptr
              sti                           ; put address into vector

;             move clptr to end of current argument
              ld    clptr
getchar:      ldi                           ; get next char in command line
              jz    endarg                  ; ja   if null char
              sub   blank
              jz    endarg                  ; ja   if space
              ld    clptr                   ; move command line ptr to next char
              add   @1
              st    clptr
              ja    getchar

;             terminate argument with null character
endarg:       ldc   0
              push
              ld    clptr                   ; clptr points to where null char goes
```

(continued)

FIGURE 10.36 (continued)

```
              sti
;             increment count in argc
              ld    argc
              add   @1
              st    argc
;             prepare for next argument
;             move vectorptr to next slot in vector
              ld    vectorptr
              add   @1
              st    vectorptr
;             move clptr to beginning of next arg
nextarg:      ld    clptr
              add   @1
              st    clptr
              ldi
              sub   blank
              jz    nextarg               ; move over blanks
              ja    getarg                ; now process next arg
; ==========================================================================
;             pass argv (the address of vector) and argc args to main
alldone:      ldc   vector                ; push address of vector
              push
              ld    argc                  ; push number of args
              push
              call main
              dloc 2                      ; deallocate parameters
              st    retcode               ; save return code from main
; ==========================================================================
;             final housekeeping code
;             check if word at loc0 still has 'Z'
testloc0:     ld     loc0
              sub    z
              jz     done                 ; if still there, ja to done
              ld    testloc0              ; start-up code at loc 0?
              jz    atloc0                ; if yes, display null ptr message
              ldc   errmsg2               ; if not, display other message
```

(continued)

FIGURE 10.36 (continued)

```
            ja    outmsg
atloc0:     ldc   errmsg3
outmsg:     sout

done:       ld    retcode          ; restore ret code from main
            halt                   ; return to op sys (sim)
;------------------------------------------------------------------
;           constants and variables
@1:         dw    1
@20:        dw    20
clptr:      dw    3900             ; address of command line
vectorptr:  dw    vector           ; array of char ptrs to the arguments
blank:      dw    ' '
argc:       dw    0                ; count of the number of arguments
vector:     dw    20 dup 0         ; space for 20 arg pointers
z:          dw    'Z'
retcode:    dw    0
errmsg1:    dw    "\nToo many command line arguments\n"
errmsg2:    dw    "\nStart-up code corrupted\n"
errmsg3:    dw    "\nNull pointer assignment\n"
            extern main
            end start_up
```

Here is a program that accesses the command line arguments. Suppose its executable file is kangaroo.mac.

FIGURE 10.38

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  cout << "argc    = " << argc << endl;
  cout << "argv[0] = " << argv[0] << endl;
  cout << "argv[1] = " << argv[1] << endl;
}
```

Let's now run the executable file `kangaroo.mac` with

```
sim /z kangaroo hello
```

The `/z` argument disables `sim`'s debugger. `sim`

1. Loads `kangaroo.mac`.
2. Places the command line (excluding `sim` itself and any argument that starts with '/' or '-') into main memory starting at location 3900 decimal
3. Passes control to start-up code within `kangaroo.mac` (because of the `end` statement in start-up code).

Start-up code creates and passes the `argc` and `argv` parameters to the `main` function in `kangaroo.mac`. The `main` function then outputs

```
argc    = 2
argv[0] = kangaroo
argv[1] = hello
```

FIGURE 10.39

```
main:       ldc   @m0           ; cout << "argc = << argc << endl;
            sout
            ldr   1
            dout
            ldc   '\n'
            aout

            ldc   @m1           ; cout << "argv[0] = "<< argv[0] << endl;
            sout
            ldr   2
            ldi
            sout
            ldc   '\n'
            aout

            ldc   @m2           ; cout << "argv[1] = "<< argv[1] << endl;
            sout
            ldr   2
            add   @1
            ldi
            sout
            ldc   '\n'
            aout

            ldc   0
            ret
@m0:        dw    "argc    = "
@m1:        dw    "argv[0] = "
@m2:        dw    "argv[1] = "
@1:         dw    1
            public main
```

Why main should return a valid error code.

FIGURE 10.40

UNIX

s1

```
p1
p2
```

s2

```
if p1
then
p2
fi
```

Microsoft Windows

b1.bat

```
p1
p2
```

b2.bat

```
p1
if errorlevel 1 goto done
p2
:done
```

What is the effect of the **static** keyword on a global variable declaration and on a function definition?

Answer: restricts scope to the file.

FIGURE 10.41

```
// module 1
void f1();
void f2();
extern int gv;
extern int sgv;
int main()
{
    gv = 0;
    f2();
    sgv = 1;  // link error
    f1();     // link error
    return 0;
}
```

```
// module 2
int gv;
static int sgv;
static void f1()
{
    int lv;
    static int slv;
    gv = 2;
    sgv = 3;
    lv = 4;
    slv = 5;
}
void f2()
{
    gv = 6;
    sgv = 7;
    f1();
}
```

How does **static** restrict scope?

Answer: by suppressing the generation of a public directive.

FIGURE 10.42

```
;           module 1
main:       ldc   0          ; gv = 0;
            st    gv

            call @f2$v       ; f2();

            ldc   1          ; sgv = 1;
            st    sgv

            call @f1$v       ; f1();

            ldc              0
            ret
            public main
            extern gv
            extern sgv
            extern @f2$v
            extern @f1$v
```

```
;           module 2
@f1$v:      aloc 1           ; allocate lv

            ldc   2          ; gv = 2;
            st    gv

            ldc   3          ; sgv = 2;
            st    sgv

            ldc   4          ; lv = 4;
            str   0

            ldc   5          ; slv = 5;
            st    @s0_slv

            dloc 1           ; dealloc lv
            ret
gv:         dw    0
sgv:        dw    0
@s0_slv:    dw    0
            public gv
;
@f2$v:      ldc   6          ; gv = 6;
            st    gv

            ldc   7          ; sgv = 2;
            st    sgv

            call @f1$v       ; f1();

            ret
            public @f2$v
```

It is good that in C++ variables that are not locally defined are *not* assumed to be external variables—that is, they are not assumed to be global variables defined in another module.

See line 10 in the next slide.

C++ requires a prototype or function definition preceding a call. Thus, the C++ compiler detects the error on line 11. The C compiler does not. Both compilers detect the error on line 10.

FIGURE 10.43

```
1     void test()
2   {
3       .
4       .
5       .
6   }
7   int main()
8   {
9      int temp;
10     temmp = 22;    // misspelling of variable temp
11     tesst();       // misspelling of function test
12      .
13      .
14      .
15     return 0;
16  }
```